

An Agile Approach to Modernisation

[Principles and practices to de-risk the modernisation process]

ThoughtWorks®

September 2009

[Jon Pither]

"Enterprises will have twenty five to thirty percent of their employees with legacy skills eligible to retire in the next three years"

"...a less than successful migration can leave a department with an increased and unnecessary burden of complexity, less well placed to meet the demands of the business"

INTRODUCTION

Today in the IT industry we are seeing a wave of CIOs and other stakeholders looking to modernise their technology platforms. Reasons for doing so include the need for replacing mission-critical systems that are becoming prohibitively expensive to change, and of needing to reduce the amount of technologies in a portfolio that are simply becoming obsolete; a report¹ by Gartner in 2008 states that 'enterprises will have twenty-five to thirty percent of their employees with legacy skills eligible to retire in the next three years'. Gartner also makes the prediction that by the end of 2010, 'more than one-third of all application projects will be driven by the need to deal with technology or skills obsolescence.'²

One widespread example of an organisation facing IT modernisation is that of the mainframe. A white paper from Microsoft³ states that 'organizations with mainframe-based systems are spending about 75 percent of their development resources simply to maintain existing applications', and claim that 'the people who know mainframe technology are steadily retiring'.

Clearly, modernising a legacy technology stack is likely to represent a large scale change for many IT departments, and one that will incorporate a great deal of risk to the wider business. As software consultants we are seeing that a less than successful migration can leave a department with an increased and unnecessary burden of complexity, less well placed to meet the demands of the business, and to be faced with a previously unanticipated demand for extra resources (and possibly the need for an additional modernisation!).

In this article I'll be highlighting a set of principals and practices drawing from the Agile philosophy that should help to steer an organisation around some of the hazards that may be present on the 'modernisation' path.

"In order to judge whether any large-scale initiative has been successful we need to know beforehand what the 'success criteria' really is"

"Typically, modernisation objectives will reflect needs such as wanting a simpler technical architecture that will allow developers to 'go faster'"

PRACTICE: STATE THE BUSINESS OBJECTIVES

In order to judge whether any large-scale initiative has been successful we need to know beforehand what the 'success criteria' really is. In the case of modernisation, will it be when the desired business functionality has been moved entirely across onto a new platform? Or will it be when the legacy platform is still operational but serving up only a targeted subset of its previous responsibilities.

In the pre-mentioned scenario of a mainframe modernisation this could mean the difference between shutting the mainframe down, and the continued usage of it but in a lesser role (such as retaining the mainframe doing bulk intensive data processing for which it's suited, while the traditional 'green screens' are replaced by new front ends written in a modern language such as Java or .Net). These two approaches could differ widely in terms of time and cost.

Knowing upfront what the *business objectives* are is a starting point for forming measurable success criteria that can be used for a modernisation agenda. Typically, modernisation objectives will reflect needs such as wanting a simpler technical architecture that will allow developers to 'go faster', of wanting to attract a wider pool of talent from the job market, or to ensure that increasingly challenging 'non-functional requirements' such as scaling, security, and third party integrations can be met.

A workshop involving brainstorming sessions with key business and technical stakeholders is a useful mechanism for capturing what needs the business has of its IT infrastructure. Once the objectives have been made explicit, we are then given the basis to ensure that all subsequent actions carried out under the banner of 'modernisation' can be tied back to what business actually wants. On some projects where there is a lot of activity but with question marks over whether all of it is completely necessary, having the business objectives clearly mapped out allows us to take a step back and to measure where all the current in-flight actions sit in terms of importance (e.g. business objectives in one column, current actions in the other).

Lastly, once we have our business objectives clearly marked out, having people with business ownership and insight co-located and involved with the software development teams on an ongoing basis⁴ is one of the best ways to ensure that what gets developed is of high business priority.

“one option for modernisation is to treat a legacy system as a black box ripe for a like-for-like replacement”

“there is the risk that developers will get bogged down in replicating in the new system some complex functionality that the business doesn't really care about, and may not actually be needed”

Stories allow for development efforts to focus first on what a system *should* do, not what it *did* do

PRACTICE #2: KEEP THE SCOPE REIGNED IN

Now that 'modernisation' has been defined in terms of some solid success criteria that should at least help prevent it from turning into some kind of white elephant, we must now look at ways of ensuring that the project does not get off-track once all the wheels are in motion.

As previously touched upon, one option for modernisation is to treat a legacy system as a black box ripe for a like-for-like replacement. Trivially, in this scenario the commandment passed down from above to the implementation team would be to install a new system that should do pretty much the same as what the old one did; to replicate across the same system behaviour and to serve pre-existing, supposedly well understood business requirements.

In this traditional scenario there is no real mechanism in place to ensure that what gets developed is consistently of high business value. Consequently with the business having effectively taken a step back, there is the risk that developers will get bogged down in replicating in the new system some complex functionality that the business doesn't really care about, and may not actually be needed (and so forth raising the possibility of making any new system 'bug-for-bug' compatible⁵ with the old one).

A technique we can use to keep development on track is to make use of 'Stories'⁶, a practice originating from the Agile software methodology 'Extreme Programming'⁷. Simply stated, 'stories' are meant to act as placeholders for bite-sized requirements, with each story representing an independent slice of system behaviour that can be individually prioritised by the business, developed and tested. The INVEST⁸ acronym is a good reference point for describing how a story should ideally be formed (**I**ndependent, **N**egotiable, **V**aluable, **E**stimable, **S**mall and **T**estable).

Stories allow for development efforts to focus first on what a system *should* do, not what it *did* do, in terms of what the business wants. It is this approach that helps a development team avoid having to reverse-engineer an older system as to come up with requirements for a new one. Additionally, having clear requirements in such an easily accessible and modifiable format also reduces the need for large up-front requirement documents (which are traditionally sent back and forth endlessly as changes are requested, taking up large amounts of time).

“stories... give us a virtual 'safety net' to ensure that the scope of the modernisation does not exceed what is thought to be achievable given a suitable time scale”

“Agile processes, such as breaking the business requirements into discreet stories... along with iterations and retrospectives... help ensure that development stays on-track.”

Looking at the wider process, ‘stories’ can also give us a virtual 'safety net' to ensure that the scope of the modernisation does not exceed what is thought to be achievable given a suitable time scale.

We get this when we assign numerical values to the stories based on their individual estimated complexity, and then by tracking our progress through them. For example if we assign 1 point for a small story, 2 for a medium, and 3 for a hard, and then if we work out that we're doing x amount of points per week, then we should be given some idea of how much time we need to reach any particular milestone (a milestone simply being a set of stories)⁹.

In addition, we can utilise the concept of 'iterations' to add in some more such safety nets. By carrying out development in time-boxed cycles – for example a single iteration lasting between one and four weeks – the team is immediately given a routine to work around. This opens up the opportunity for the business to check-in on progress, to see what work has been done, and to make adjustments as necessary. The use of ‘retrospectives’ at the end of the iteration is a practice where the team as a whole (business stakeholders included) get to stand back and to reflect upon how things are going, and to think of what could be improved (thus bringing about a real sense of continuous improvement).

Modernisation is a risky endeavour that once off course can cost an organisation a large amount of time, effort and money (labelled by some as the so-called ‘death march’¹⁰). Agile processes, such as breaking the business requirements into discreet stories that can be individually prioritised and tracked, along with iterations and retrospectives, gives the team a series of safety nets to help ensure that development stays on-track.

"we need to be sure that we're not operating underneath a 'silver-bullet' mentality"

"there's a real risk that what was holding an organisation back from achieving its objectives in the past will continue to impede progress in the future"

"Having people thinking in terms eliminating waste is a useful way of getting them to focus on current problems ahead of what a future state may offer"

PRACTICE #3: AVOID THE SILVER BULLET MENTALITY

Modernisation is a large-scale change initiative that should solve a series of problems faced by the business. It's important that from the outset we are able to discern between which problems exist that will be solved by the change, and of those that will not be. In other words, we need to be sure that we're not operating underneath a 'silver-bullet'¹¹ mentality.

If we take an example business objective of 'reducing the number of bugs', we are obligated to understand what is happening that is allowing these insidious bugs to fester in the first place; the root cause. A high number of 'defects' in the software could well be traced back to an unwieldy system architecture ripe for replacement, but they could also largely originate from an unclear requirements-gathering process, or from the 'overproduction' of requirements that are not very high up on the users' priority list.

These latter concerns are side effects of an incumbent software development process, and do not stem directly from the technology platform. It may be that modernising technology significantly reduces a number these problems and does indeed allow the business to 'go-faster', but there's a real risk that what was holding an organisation back from achieving its objectives in the past will continue to impede progress in the future.

A way of tackling this is to make explicit from the start what an organisation's current problem areas are, so that we can better set expectations of what modernisation will be able to achieve. The Lean manufacturing process places a strong emphasis on the notion of 'waste' (expenditure of resources not resulting in value for the end customer), where it defines waste into seven categories (such as defects, waiting, and over-production)¹². Having people thinking in terms eliminating waste is a useful way of getting them to focus on current problems ahead of what a future state may offer. Holding a waste discovery workshop with the waste-categories written up on a flipchart as fodder for thought can be a good start.

Another Lean tool (called a 'Value Stream Mapping'¹³) is to map out the various processes and steps that constitute a software development pipeline, along with the addition of metrics such as how long each step takes. The goal here is to track down where waste is occurring and what bottlenecks exist, hopefully resulting in a greater impetus to tackle these problems.

“We may even find going forward that priorities shift entirely onto rectifying problems in the current process, ahead of any large-scale technological change”

“As a system is modernised it should ideally be simplified”

“the more moving parts there... the riskier it will become to change”

By getting a handle on what problems modernisation will achieve and of those that it will not, we are consequently able to set expectations to the wider business that there really is no ‘silver bullet’. We may even find going forward that priorities shift entirely onto rectifying problems in the current process, ahead of any large-scale technological change.

PRACTICE #4: GUARDING AGAINST COMPLEXITY

As a system is modernised it should ideally be simplified. If the overall complexity is increased then the cost of software development and maintenance will go up, and in the long term it will be more difficult for people to understand the 'bigger picture' of exactly what is going on. Needless to say, the 'expanded' system will become more expensive and riskier to change.

This risk is likely to be realised in the scenario where an existing system is *extended* with a new platform capable of delivering some of the stated success criteria (as oppose to a total 'big bang' replacement). It should be noted that this approach might be unavoidable, especially if duties of the old system need to be transferred to a new platform on a piecemeal basis.

For purposes of illustration we can turn back to the previously used mainframe example, and consider what would happen if indeed the existing green screens were to be replaced with a new user-interface based on more modern technologies.

Looking at the mainframe, there would be an existing set of processes around building and deploying mainframe compatible code (such as COBOL) from development right through to the live environment. Since we're now adding a new set of 'front-end' technologies into the mix, then we'll also have to introduce a set of corresponding deployment processes that mirror the ones just described. These additional processes will have to be planned for and managed like any other, and will each come with their own individual cost (along with an additional cost of 'syncing' the processes up, in order to get a combined release out into production).

This is complexity that comes large in scope and can be difficult to get to grips with; the more moving parts there are in such a deployment, the riskier it will become to change. Other examples of increased complexity include the management of two entirely different code-bases and source repositories, and of integrating a varying suite of systems to talk to one another, across what could be a wide set of integration points.

“the use of stories and iterations gives a team a series of safety nets to ensure that work does not get off track”

“we can make use of techniques such as metrics to give us a real sense of the impact complexity is having on a project”

“The idea is that once such ‘technical debt’ has been made explicit, teams will seek to reduce it”

Fortunately, there are some Agile practices that can be used to help keep burdening complexity under control, or least to make it more transparent. As discussed earlier, the use of stories and iterations gives a team a series of safety nets to ensure that work does not get off track, as it can have the propensity to do so when complexity starts to spiral. The use of frequent retrospectives gives the team members the opportunity to reflect on any areas of waste that may be festering, and to spot complexity as it arises.

In addition, similarly to how we’re able to scrutinise a software process for waste, we can make use of techniques such as metrics to give us a real sense of the impact complexity is having on a project. Back to our example of the overall build and deployment process becoming more convoluted, having metrics around each step of the pipeline will help flag any adverse slow-downs a technical migration is having. If we know that before in yesterday's world it used to take one hour to deploy our changes into the QA environment, and now in today's world it takes three, then we can clearly see that a problem has arisen.

A different tool that can be used to make complexity more explicit is the use of the metaphor 'technical debt'¹⁴. Technical debt exists when a design choice has been made in the present that creates more work for us in the future (hence the notion of debt leading to interest payments). Teams often choose to tackle technical debt by writing up on a dedicated wall space what items of technical debt a system has. Typically, items of debt will refer to areas of complexity in the system, as well as other issues such as technical ‘todos’, and areas of questionable code quality. The idea is that once such ‘technical debt’ has been made explicit, teams will seek to reduce it.

Modernisation should tend towards reducing complexity throughout a system. By measuring the impact it is having through the use of metrics, and by making it explicit through the metaphor of technical debt, we can get a better handle on complexity as it arises. By using previously mentioned safety-net techniques such as stories, iterations and retrospectives, we can help to ensure that it doesn’t spiral.

"A well-trodden route to take towards modernisation is to buy big 'middleware' software products"

"some software-vendors will want to sell as many products as they can"

"a sledgehammer has been purchased to crack open a nut"

PRACTICE #5: GUARD AGAINST BLOATWARE

Choosing what the future technical platform will be is one of the key strategic decisions to make in a modernisation process. The repercussions are far reaching, impacting not only hardware decisions and the hiring process for targeted skill-sets, but also the nature of the business requirements themselves (for example one might hope that a new platform would offer some new unforeseen business opportunities).

A well-trodden route to take towards modernisation is to buy big 'middleware' software products that come with an in-depth feature-set, a comprehensive support capability (albeit at a cost), good documentation, and with an architecture that should cater for scalability, expansion and integration. Example categories of 'big' products include ESBs (Enterprise Service Buses), B2B (business to business) integration platforms, and BPM (Business Process Management) tools.

Products falling into these categories can often justify the big up-front spend that they usually require. For example, having a product that meets its Service Level Agreement in terms of scalability and performance that is crucial to the business, such as a messaging queuing system where the messages are absolutely guaranteed to be delivered across continents 100% of the time, can readily provide a return on investment.

On the other hand, we also need to consider that some software-vendors will want to sell as many products as they can, and not always to where they are most needed. As software consultants, we are frequently brought in to advise on software architectures that are built around heavyweight products of which their inherent complexity far outstrips their necessity. It is for these types of products that the term "Bloatware" was fashioned.

In our scenario of mainframe modernisation, one example of 'bloatware' could be the customer purchasing a large and expensive web application server when all that they really want to do is to have a few web pages directing the traffic data back towards the mainframe for processing purposes. Here, a sledgehammer has been purchased to crack open a nut, and the customer now faces the arduous task of supporting a much larger product than they really needed. Add in some more products (a complex workflow engine, a data-warehousing module) and before long the problem of spiralling complexity will again rear its head.

““Vendor lock-in” . . . is often a root-cause for modernisation in the first place”

“92% of open source adopters gained their confidence from met or exceeded expectations around software quality”

The “last responsible moment”, the idea being that we should defer making large decisions such as purchasing products until we know that we definitely need them”

Aside from buying sledgehammers for the business of nut cracking, bloatware can also threaten “Vendor lock-in”, a situation that along with the perils of complexity is often a root-cause for modernisation in the first place.

'Vendor lock-in'¹⁵ occurs when an organisation is tied into a particular set of products and/or services from a vendor, and cannot switch to other providers without substantial costs. This could be because an entire system architecture is based around a particular product suite that uses proprietary protocols instead of open ones¹⁶, and one that offers no clear path for piece meal migration to other technologies. 'Vendor lock-in' could also be caused by and further maintained when an organisation has a long running 'strategic partnership' with a vendor, and where the organisations' technical direction is closely tied to that of the vendor's product roadmap.

An alternative approach in stark contrast to that of purchasing bloatware is to make use of lighter-weight, community-driven, open-source tools and frameworks. A reason for doing so is that many open-source products allow for rapid development without the need for a large upfront investment in terms of capital, time and effort (e.g. less of a need for a costly vendor selection process and any ensuing due diligence). In terms of maintaining software quality, a white-paper from Forrester looking at open-source adoption by enterprises reports that (from a pool of surveyed IT executives) “92 percent of respondents have had their quality expectations met or exceeded by open-source software¹⁷”.

Another Agile principal that we can use is to avoid ‘big upfront design’, in favour of a more evolutionary approach to software development. Following this path design decisions are left to the “last responsible moment¹⁸”, the idea being that we should defer making large decisions such as purchasing products until we know that we definitely need them (and sometimes it is genuinely the case that we don't). A point to note here is that deferment of such decisions should be a safe action to take, and one that does not bring the project into jeopardy.

Making use of open-source products often helps to align an organisation closer to the Agile philosophy of getting high quality working software out the door in a timely, lightweight manner. Because such tools are evolved through use by the community, and are usually built around standard protocols, they help us to avoid the trap of Vendor Lock-in. By deferring large design decisions until the “last responsible moment”, we lessen the risk of burdening an organisation with the perils of “Bloatware”.

“On modernisation projects where a large variety of skill-sets are needed . . . there is often a large challenge in ensuring that knowledge is spread around”

“For modernisation, where a large amount of new processes and technologies can be typically introduced, pairing is an effective mechanism for disseminating knowledge”

“an organisation can begin to realise the underlying Agile goal of “collective code ownership”

PRACTICE #6: LEVERAGE EXPERIENCE

Aside from technological concerns, producing working software is often a problem of getting people to work together effectively¹⁹. On modernisation projects where a large variety of skill-sets are needed (i.e. people with knowledge of the legacy systems as well as the new), there is often a large challenge in ensuring that knowledge is spread around, and that skills are transferred.

An anti-pattern we want to avoid is that of ‘knowledge silos’, a situation where only a small number of individuals have experience or know-how of a particular area of a system (such as a specialist legacy area). Not only can this slow down development as employees spend large amounts of time chasing each other around for information, there is also the risk that if ‘key’ people were to be “hit by a bus” (or rather if they just exit the company), then the cost to the organisation would be disproportionately large. This could effectively put the brakes on a modernisation process, as well as being a root cause for it in the first place (as in the case of the mainframe example where experienced employees are simply retiring, taking a large chunk of know-how out the door with them).

We can tackle the problem of knowledge silos forming by using targeted practices that facilitate knowledge sharing. One is the Agile practice of ‘Pair Programming’, where two developers tackle a story/task together at the same time (using the same keyboard). In this practice the knowledge and skills required for any single job are implicitly shared, and by regularly rotating the pairs (shaking up who pairs with whom), we can ensure that knowledge is spread around the whole team. For modernisation, where a large amount of new processes and technologies can be typically introduced, pairing is an effective mechanism for disseminating knowledge and ramping up an employee’s skill-set. It is also of use for bringing new-hires up to speed quickly as well as for training purposes (i.e. pairing juniors with seniors).

Once knowledge-sharing practices such as pair programming are in place, an organisation can begin to realise the underlying Agile goal of “collective code ownership”; an idea whereupon each developer has a robust knowledge of the system they are working with, and doesn’t have to pass work on to some other person with specialist insight. Once attained, collective code-ownership not only neatly side steps the “bus-hitting” problem, it brings forward efficiencies as more people are able to jump on a particular problem than before.

"a key principle that stands out for helping organisations leverage experience is that of 'individuals and interactions over processes and tools'"

"self-organising teams are not easy to form, and certainly we don't get them for free"

"By creating self-organising teams, an organisation can become truly agile"

Digging deeper into the Agile philosophy²⁰, a key principle that stands out for helping organisations leverage experience is that of 'individuals and interactions over processes and tools'. Essentially this is about empowering people to do the best job that they can, and could be considered a polar opposite to constraining people by means of bureaucracy.

An Agile practice that sets out to achieve this is the 'self organising team'. Rather than dictating a fine-grained process to team members (i.e. that they must do pair programming or that stories must have X amount of analysis before implementation), we can instead create space for a team to choose its own way of working, so that it can self-optimize. For example, the team itself can fine-tune how the pair-programming works (who pairs with whom, for how much time a pair works together), how stories are estimated (how much time the team thinks a story will take), and even how the teams themselves are made up (how many people should be in a team, does the team need to be split into smaller teams and along what boundaries does the split occur?).

As software consultants, we regularly find that self-organising teams have a larger sense of problem ownership by individuals, better communication, and crucially, a heightened morale. For modernisation projects where there are usually a large number of complex moving parts, having a team with the ability to re-adjust itself to the challenges it faces is crucial.

We have also discovered that self-organising teams are not easy to form, and certainly we don't get them for free. Instead there is required a blend of ingredients, namely: strong leaders within the team, individuals with experience of the Agile process (including working in an self-organising team), space from management to make mistakes and so to learn by them, forums to discuss continuous improvements such as retrospectives, and the ability to start small with a few employees and to grow organically as the team feels it's ready to do so.

Modernisation, like any other large IT project needs good people on-board. Agile practices such as pair programming ensure that knowledge is spread around the team once it is up and running, and helps to avoid the creation of 'knowledge silos' which can often be a ticking-time-bomb cause for modernisation in the first place. By creating self-organising teams, an organisation can become truly agile in that it can reshape itself to meet the ever-changing needs of a complex project.

"Spiralling complexity, development teams losing business focus, knowledge silos, use of bloatware leading to vendor lock-in, and not making the best use of the experience that we have are all hazards that may be encountered"

"through making use of the people that we have by empowering them to self-organise... we can effect a real movement of continuous improvement"

SUMMARY:

Migrating an organisation's mission critical systems away from an old technology stack to a new one can be an arduous process, and one that can incur a great deal of risk. Spiralling complexity, development teams losing business focus, knowledge silos, use of bloatware leading to vendor lock-in, and not making the best use of the experience that we have are all hazards that may be encountered.

By adopting some Agile processes such as the use of stories, iterations and retrospectives, we can layer some 'safety nets' into our efforts that will give an earlier transparency into when development is getting off-track. Before a modernisation project is under way, we should seek to make explicit the overall objectives of the business, and we should be aware of exactly what problems modernisation will solve, of the problems that will not be solved, so that the correct expectations to the wider business can be set.

By watching out for the pitfalls of 'bloatware', we can seek to avoid constraints such as 'vendor lock-in', and over-engineered solutions that may have induced the need for modernisation in the first place. Finally, through making use of the people that we have by empowering them to self-organise, and by encouraging them to share knowledge through techniques such as pair programming, we can effect a real movement of continuous improvement.

Looking to the future, we should hope that if these practices and principles are retained beyond the current modernisation agenda, then we should have gone a long way to preventing the wheel from turning again and the new platform becoming legacy itself once the dust has settled.

REFERENCES:

- 1 <http://www.gartner.com/it/page.jsp?id=611507>
- 2 <http://www.gartner.com/DisplayDocument?id=597615>
- 3 http://download.microsoft.com/download/E/9/7/E9734F87-C581-482A-AACA-2835DF48D40E/Business_Value_Legacy_Modernization.pdf
- 4 <http://www.govtech.com/pcio/638378>
- 5 <http://dictionary.reference.com/browse/bug-compatible>
- 6 http://en.wikipedia.org/wiki/User_story
- 7 <http://www.extremeprogramming.org/>
- 8 <http://xp123.com/xplor/xp0308/index.shtml>
- 9 <http://www.extremeprogramming.org/rules/velocity.html>
- 10 [http://en.wikipedia.org/wiki/Death_march_\(software_development\)](http://en.wikipedia.org/wiki/Death_march_(software_development))
- 11 http://en.wikipedia.org/wiki/No_Silver_Bullet
- 12 <http://continental-design.com/lean-manufacturing/handbook-1.html>
- 13 http://en.wikipedia.org/wiki/Value_Stream_Mapping
- 14 <http://martinfowler.com/bliki/TechnicalDebt.html>
- 15 <http://www.antipatterns.com/vendorlockin.htm>
- 16 A case in point: http://www.infoworld.com/infoworld/article/07/05/14/20FEsoabottle-9_1.html
- 17 <http://opensource.sys-con.com/node/848199>
- 18 <http://www.codinghorror.com/blog/archives/000705.html>
- 19 http://findarticles.com/p/articles/mi_hb3234/is_6_34/ai_n29145229/?tag=content;col1
- 20 <http://agilemanifesto.org>